The Finer Points of UVM: Tasting Tips for the Connoisseur

John Aynsley
Doulos
Church Hatch, 22 Market Place
Ringwood, United Kingdom
+44 1425 471223
john.aynsley@doulos.com

ABSTRACT

UVM, the Universal Verification Methodology for SystemVerilog, has been with us for several years now and is being increasingly adopted due to its strength as a multi-vendor standard, encouraging a consistent and re-usable approach to creating verification IP. As is often the case with such standards, there are many users who dip their toes in the water but never seem to find time to explore the full potential of UVM. This paper explores some of the finer points of UVM, building upon experience gained at Doulos from delivering training and working with engineers using UVM in a practical industrial environment.

This paper covers a lot of ground. The goal is to help engineers to progress beyond the basics of UVM by pointing to the areas of UVM that are worthy of further attention. Rather than just restating the contents of the UVM class documentation, the approach is to give some tips as to when, why, and how to use some of the deeper features of the UVM base class library.

Keywords

SystemVerilog, UVM, functional verification, constrained random verification, programming language

1. INTRODUCTION

UVM, the Universal Verification Methodology for SystemVerilog, consists of a SystemVerilog base class library, a Class Reference [2], a User’s Guide [3], and a few examples, all of which are freely downloadable [4]. According to the Class Reference, "The UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog."

The UVM class library is very rich in content. Introductory seminars or webinars on UVM do little more than scratch the surface. Formal hands-on UVM training courses typically run for 3 or 4 days of very intense training, but even that is often insufficient to teach all the detail. The goal of this paper is to pick a few of the more interesting features of UVM and dig a little deeper than would usually be done in an introductory seminar.

We start by investigating the sequencer, which is the engine used in UVM to generate stimulus. We show how to manipulate the behavior of multiple concurrent sequences with confidence and to have one sequence interrupt another, which is a particularly useful technique for creating virtual sequences where a sequence running on one sequencer is able to start and stop sequences on a different sequencer. We also point out pitfalls to avoid with concurrent sequences, such as being sure to define the parent-child relationship between sequences and to set the proper sequence arbitration algorithm.

Continuing with the theme of sequences, we investigate the sequence library. Both the current UVM release and several of its ancestors have included trial implementations of a so-called sequence library, which attempts to provide a mechanism for packaging and re-using a set of related sequences.

UVM puts a lot of emphasis on sequences for the creation of modular, reusable stimulus generation, and allows sequences to be related in a number of ways, including nested, virtual, and layered sequences. The use of layered sequences poses questions as to the best way to structure those sequences for reusability. This paper explores how to minimize unwanted dependencies between layered sequences when creating so-called translation sequences, which translate between layers in a protocol stack. This paper also explores the options for passing request and response information up and down the stack of layered sequences, including the concepts of request and response ids and response queues, and contrasts this with the range of more ad hoc methods for communication between components also available in UVM.

The joint concepts of a resource database and a configuration database were introduced with UVM 1.0 to extend and generalize the concept of configuration tables from OVM. The resource database is a general container for shared resources, where each resource has a scope, a name and a typed value. The configuration database uses the resource database by interpreting the scope of each resource as a hierarchical name in the UVM component hierarchy. UVM improves upon OVM by allowing values of arbitrary type to be stored directly in the database without the need for wrappers, but at the cost of some complexity for the user in dealing with the twin concepts of the resource and configuration databases. The ability to use wildcards in the scope string and to make multiple identical entries in the database can be very powerful mechanisms when properly understood, and are explored in this paper.

The configuration database and the reporting mechanism in UVM are linked to the UVM component hierarchy, though the programming interface to both these mechanisms now permit either to be called from sequences, despite the fact that sequences are not part of the component hierarchy. This paper exposes some pitfalls when accessing the configuration database from sequences, and makes some recommendations on best practice.

2. ASSUMPTIONS

This paper is based on the Accellera Systems Initiative Universal Verification Methodology version 1.1c, which was released in October 2012.

This paper assumes the reader has a certain level of familiarity with the SystemVerilog language and with the UVM class library. In particular, a familiarity with the following UVM concepts is assumed: the component hierarchy, the factory, the configuration database, the phasing mechanism, transaction-level communication, analysis ports, agents, sequencers, sequences, virtual sequences, tests, and objections.

3. THE SEQUENCER

The UVM sequencer is the engine for running sequences. Each sequence runs on a sequencer, in much the same way as a computer program runs on a processor. Like a processor, a sequencer is static and has fixed connections to other components in the UVM verification environment. Like a computer program, a sequence is dynamic, has a start and an end point in time, and potentially has to compete with other sequences trying to run on the same sequencer at the same time. Each sequence has a task, named **body**, which may generate transactions itself or may run other nested sequences to do so. Those nested sequences may in turn generate transactions of the same type as their parent sequence or may run other nested sequences, and so on to any depth. In the end, each regular sequence generates a single stream of transactions of a given type, though it may call other nested sequences to do its work. An exception to this rule is the so-called *virtual sequence*, which may execute several nested sequences across multiple separate sequencers, each of which may individually generate transactions of different types.

It is often sufficient to use the uvm_sequencer out-of-the-box without the need to extend the uvm_sequencer base class. So the following code is often sufficient to create a sequencer:

```
typedef uvm_sequencer #(my_tx) my_seqr;
...
my_seqr seqr;
...
seqr = my_seqr::type_id::create("seqr", this);
```

A simple user-defined sequence class might look like this:

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

  // Boiler-plate constructor code
  function new(string name = "");
    super.new(name);
  endfunction: new

  // The body task does the work of the sequence
  task body;
    repeat(4)
    begin
      // Create a new transaction object
      req = my_tx::type_id::create("req");

      // start_item waits for the driver
      start_item(req);

      if (!req.randomize())
        `uvm_error("", "failed to randomize")

      // finish_item sends the request to the driver
      finish_item(req);
    end
  endtask
endclass
```

The sequence above generates 4 random transactions of type my_tx. If the sequence above were to be called from another sequence, the body task of the parent sequence might look as follows:

```
class top_seq extends uvm_sequence #(my_tx);
  ...
  `uvm_declare_p_sequencer(my_seqr)
  ...
  task body;
    repeat(3)
    begin
      my_seq seq;
      seq = my_seq::type_id::create("seq");
      if (!seq.randomize())
        `uvm_error("", "failed to randomize")
      seq.start(p_sequencer, this);
    end
  endtask
```

In the code fragment above, the sequence my_seq is called 3 times from top_seq. Sequence my_seq runs as a child sequence of top_seq and both sequences run on the same sequencer. This is where things start getting a little more subtle. my_seq is made to run on the *same sequencer* as top_seq by passing the variable **p_sequencer** as the first argument to the **start** method. **p_sequencer** is introduced into the user-defined sequence class using the macro **`uvm_declare_p_sequencer**, and points to the sequencer that the sequence is running on. This variable is useful whenever a sequence needs access to the sequencer on which it is running.

Moreover, my_seq is made to run as a child of top_seq by passing a reference to the parent sequence (**this**) as the second argument of the **start** method. It becomes critical to define the parent-child relationship between sequences in this way when it comes to controlling sequence execution from virtual sequences, as we will see below.

As the code stands above, my_seq is called 3 times *in sequence* from the parent sequence. Because the **start** method is blocking, each instance of my_seq only starts running after the previous instance has completed. However, if we allow the child sequences to run *concurrently* on the same sequencer, we open the door to lots of interesting issues:

```
task body;
  fork
    begin
      seq1 = my_seq::type_id::create("seq1");
      if (!seq1.randomize())
        `uvm_error("", "failed to randomize")
      seq1.start(p_sequencer, this);
    end
    begin
      seq2 = my_seq::type_id::create("seq2");
      if (!seq2.randomize())
        ...
      seq2.start(p_sequencer, this);
    end
    begin
      ...
      seq3.start(p_sequencer, this);
    end
  join
endtask
```

You are encouraged to try this example for yourself. What you will find is that transactions from the 3 sequences (seq1, seq2, seq3) are strictly interleaved. This is no accident but is a deliberate feature of the UVM sequencer, and can be brought under user control.

3.1. The arbitration queue

Each sequencer has an arbitration queue containing references to all the sequences that are trying to run on the sequencer at the current point in time (seq1, seq2, seq3 in the example above). Each sequencer also has an arbitration algorithm used to select the next sequence item from the queue. The default arbitration algorithm is FIFO such that the first sequence to get started gets served first and, having generated one transaction, gets sent to the back of queue. The result is that when multiple sequences are competing to run on the same sequencer, they get scheduled in round-robin order so that the transactions are strictly interleaved.

The arbitration algorithm used by each sequencer can be selected by the user from a set of built-in algorithms or can be user-defined. Several, but not all, of the built-in algorithms make use of the priority of the sequence. It turns out that the default FIFO algorithm *ignores* the sequence priority, which can be very confusing for newcomers trying to understand the behavior of their code.

Here is an example of selecting a built-in arbitration algorithm, which in this case does make use of the sequence priority:

```
task body;
  p_sequencer.set_arbitration(
                      SEQ_ARB_STRICT_RANDOM);
  fork
    begin
      seq1 = my_seq::type_id::create("seq1");
      if (!seq1.randomize())
        `uvm_error("", "failed to randomize")
      seq1.start(p_sequencer, this, 1);
    end
    begin
      ...
      seq2.start(p_sequencer, this, 2);
    end
    begin
      ...
      seq3.start(p_sequencer, this, 3);
    end
  join
endtask
```

The algorithm SEQ_ARB_STRICT_RANDOM strictly selects sequences with a higher priority (larger integer) before sequences with a lower priority (smaller integer), and in the case of sequences with equal priority makes a selection at random. In the example above, this would result in seq3 running to completion before seq2 is allowed to generate its first transaction. The sequence priority can be set by being passed as the third argument to the **start** method, or by calling the **set_priority** method of the sequence object.

In order to provide a user-defined algorithm it is necessary to override the **user_priority_arbitration** method of the sequencer and to select the SEQ_ARB_USER algorithm. This requires a user-defined sequencer class, for example:

```
class my_sequencer
                 extends uvm_sequencer #(my_tx);
  ...
  function integer user_priority_arbitration(
                   integer avail_sequences[$]);
    foreach (avail_sequences[i])
    begin
      integer index         = avail_sequences[i];
      uvm_sequence_request req =
                      arb_sequence_q[index];
      int pri               = req.item_priority;
      uvm_sequence_base seq = req.sequence_ptr;

      if (pri > max_pri)
      begin
        max_pri   = pri;
        max_index = index;
      end
    end
    return max_index;
  endfunction

endclass
```

The **user_priority_arbitration** method is passed a queue containing indexes into the arbitration queue of the sequencer, which is itself named **arb_sequence_q**. From that it is possible to retrieve the priority of each sequence and the sequence objects themselves, both of which can be used to calculate and return the index number of the next sequence to be selected. This mechanism gives the user total control over the order in which the sequencer selects which sequence to run next. The sequence priority can even be adjusted dynamically, if required.

3.2. Virtual sequences

A *virtual sequence* is a sequence that happens not to generate any transactions itself but does its work by starting child sequences on other sequencers. Virtual sequences are typically used in UVM to co-ordinate the behavior of multiple sequencers within multiple agents connected to multiple interfaces of the design-under-test.

In early versions of OVM, virtual sequences and virtual sequencers were distinguished from regular sequences and sequencers by means of separate base classes. Virtual sequences could only run on virtual sequencers. This legacy still lives on in the minds of some UVM users, though virtual sequences and sequencers have long since been collapsed into their regular brethren.

In UVM, a virtual sequence extends the same base class as a regular sequence and can run on any sequencer. In fact, because a virtual sequence is not obliged to be specialized with a specific transaction type when extending **uvm_sequence** (see the example below), and because a virtual sequence will not, by definition, call **start_item** and **finish_item**, there are fewer restrictions on the choice of sequencer. A virtual sequence can run on its own dedicated sequencer, can run on a sequencer used to run regular (non-virtual) sequences, or can even run on the *null sequencer* (explained below). Aside from the desire to group together related sequences to facilitate reuse, the choice would be made on the basis of whether the user needs to have the virtual sequence access properties of an existing sequencer.

When running any sequence, the **start** method of the sequence calls its **body** method, which may in turn start child sequences, and so on until a child sequence attempts to generate a transaction, at which point the entire call stack will be stalled until the downstream component (typically a driver) requests a transaction. The priority of the sequence, passed as an argument to **start**, will be used as the default priority for any child sequences, and ultimately as the default priority of any transactions. Priority and arbitration are ultimately only relevant to the transactions generated by the sequences, not to the sequences themselves, so the sequencer arbitration queue, as described above, is not directly relevant to virtual sequences but only to the transactions generated by their children. This is why it is possible to start a virtual sequence on the null sequencer, for example:

```
virtual_seq.start(null, this, priority);
```

It is meaningful to set the priority of a virtual sequence, even when that virtual sequence is running on the null sequencer, because the priority will be inherited by the children of the virtual sequence and hence by their transactions.

3.3. lock and grab

There is another important mechanism that can be used to control the order of sequence execution, namely, the ability of a sequence to "lock" or gain exclusive control over a sequencer. Once a sequence has locked a sequencer, only that sequence can have its sequence items executed on the sequencer: all other sequence items in the arbitration queue will be bypassed until the lock is released. The sequence lock mechanism is of particular interest when writing virtual sequences.

Imagine a UVM sequencer within an agent connected to the DUT. That sequencer may be generating background traffic appropriate to the particular interface it is connected to. A virtual sequence, which co-ordinates multiple agents, may want to take control of that low-level sequencer to inject some specific transactions or to handle an interrupt. This can be accomplished using the sequence lock mechanism.

Here is an example of a virtual sequence:

```
class virtual_seq extends uvm_sequence;
  `uvm_object_utils(virtual_seq)
  my_sequencer seqr; // Reference to another sequencer
  ...
  task body;
    my_seq seq;
    seq = my_seq::type_id::create();
    seq.starting_phase = starting_phase;
    if (!seq.randomize()) ...

    // Take exclusive control of another sequencer
    this.lock(seqr);

    // Run sequence on that sequencer
    seq.start(seqr, this);

    // Relinquish control
    this.unlock(seqr);
    ...
```

The call **this.lock(seqr)** gives the calling sequence virtual_seq exclusive access to the sequencer **seqr**. Assuming **seqr** is not already locked by some other sequence, the virtual sequence above will be able to run its own sequence before giving up the lock. On the other hand, if the sequencer is already locked, the call to **lock** above will block until the sequencer becomes available by being unlocked elsewhere. By definition, only one sequence can lock a given sequencer at any given time, and any other calls to **lock** get sent to the back of the queue.

There is an alternative method **grab** that is identical in effect to **lock** except that in the event that the attempt to lock the sequencer is not immediately successful, the pending request gets sent to the front of the arbitration queue rather than to the back. This gives the user a certain degree of control over the behavior in the event that multiple sequences attempt to lock the same sequencer simultaneously. Once a call to **lock** or **grab** has taken control of a sequencer, the owner cannot be interrupted by any other call to **lock** or **grab** until it has explicitly relinquished control by calling **unlock** or **ungrab**. **lock** is polite and goes to the back of the queue, whereas **grab** barges in at the front of the queue. However, neither **lock** nor **grab** requests are affected by the arbitration algorithm or sequence priority: both are serviced ahead of any regular sequence items in the arbitration queue. In other words, the distinction between **lock** and **grab** is only important with respect to the order in which concurrent locks and grabs are serviced on the same sequencer.

4. THE SEQUENCE LIBRARY

The concept of a sequence library has been around since the days of URM and AVM, prior to their merging into OVM. UVM contains a prototypical sequence library implementation, though at the time of writing the uvm_sequence_library is still not included in the official UVM documentation. Note that OVM included a set of macros for creating a "sequence library" which have been deprecated in UVM. The uvm_sequence_library being discussed here is not the same as the deprecated sequence library mechanism from OVM.

The idea behind the sequence library is to have a library of sequences (naturally enough) where the sequences get run in turn, one-at-a time. This is in contrast to the discussion on the arbitration queue above, where we were considering the issue of what happens when several sequences attempt to run in parallel on the same sequencer: the sequence library runs several sequences in series on the same sequencer.

The sequence library should *not* be used as the mechanism for managing and starting every sequence: the techniques for starting sequences as discussed in previous sections are sufficient for many purposes. The sequence library is a specific mechanism for a specific purpose, that is, to identify a set of sequences and then be able to control the order in which those sequences execute either by selecting a random execution order or by providing a user-defined algorithm.

In use, a sequence library looks like a fancy sequence (class uvm_sequence_library extends uvm_sequence):

```
class my_seq_lib
          extends uvm_sequence_library #(my_tx);
  `uvm_object_utils(my_seq_lib)
  `uvm_sequence_library_utils(my_seq_lib)

  function new(string name = "");
    super.new(name);
    init_sequence_library();
  endfunction
endclass
```

Notice that the macro uvm_sequence_library_utils and the function init_sequence_library must be called when defining a sequence library, and that the user does not supply a **body** task for the sequence library: the **body** task is built into the base class.

You start a sequence library on a sequencer as you would an ordinary sequence, except that you first add sequences to the library and set bounds on how many sequences will be run. This might be done from the run phase of a test, for example:

```
task run_phase(uvm_phase phase);

  // Create the sequence library object using the factory (as usual)
  my_seq_lib lib = my_seq_lib::type_id::create();

  // Add several user-defined sequences to the library
  lib.add_sequence( seq1::get_type() );
  lib.add_sequence( seq2::get_type() );
  ...

  // Must set sequence library properties before randomizing the library
  lib.selection_mode = UVM_SEQ_LIB_RAND;
  lib.min_random_count = 15;
  lib.max_random_count = 20;

  // Randomize sequence library object to set the number of sequences
  if ( !lib.randomize() ) ...
```

```
    // Set starting_phase because the sequence library raises an objection
    lib.starting_phase = phase;
    lib.start(m_env.m_seqr);
    ...
```

The example above creates a new sequence library object, adds several user-defined sequences to the library, sets the algorithm used to select the order in which the sequences are to be run (UVM _SEQ_LIB_RAND meaning that each sequence run is selected at random from the full set of sequences in the library), and sets minimum and maximum bounds on the number of sequences to be run. The sequence library object is then randomized and started on a specific sequencer (where it might have to compete with other sequences running concurrently on the same sequencer, as discussed above).

Instead of adding sequences to a sequence library object, it is also possible to add sequences to the sequence library *class* such that they are available to all instances of that sequence library. This is easy to do by calling a static method from outside the class, as follows:

```
  my_seq_lib::add_typewide_sequence(
                              seq3::get_type() );
  my_seq_lib::add_typewide_sequence(
                              seq4::get_type() );
  ...
```

In the example above, the properties of the sequence library are set procedurally after the sequence library object has been instantiated by the factory. As you would expect in UVM, it is also possible to set these properties in advance of the creation of the object by using the configuration database. To make this possible, you also need to select the sequence library as the default sequence for the run phase of a particular sequencer, for example:

```
uvm_config_db #(uvm_object_wrapper)::set(
    null, "*.m_seqr.run_phase",
    "default_sequence", my_seq_lib::get_type() );

uvm_config_db #(int unsigned)::set(
    null, "*.m_seqr.run_phase",
    "default_sequence.min_random_count", 15 );
```

As regards the order in which the sequences within the library are chosen for execution, you can have as much or as little control as you want. For the ultimate in flexibility you could select a user-defined algorithm, for example:

```
   class my_seq_lib
          extends uvm_sequence_library #(my_tx);
     ...
     // Override the built-in select_sequence method
     function int unsigned select_sequence(
                                 int unsigned max);
         static int unsigned counter;
         select_sequence = counter;
         counter++;
         if (counter > max)
           counter = 0;
     endfunction
   endclass

   ...
   lib.selection_mode = UVM_SEQ_LIB_USER;
```

The select_sequence method is required to return an integer in the range 0 to max, inclusive. In the current prototypical implementation of the sequence library (UVM-1.1c), the default implementation of select_sequence actually returns an integer in the range 0 to max-1, so cycles through one-too-few sequences. The example above shows a user-defined select_sequence method that fixes this bug.

In selecting which sequence to execute next from the library, you may sometimes require information about which sequence is which. This can be achieved by calling the get_sequences method, as follows:

```
   function int unsigned select_sequence(
                              int unsigned max);
     uvm_object_wrapper seqq[$];
     get_sequences(seqq);
     foreach (seqq[i])
       if (seqq[i] == seq1::get_type())
         ...
       else if (seqq[i] == seq2::get_type())
         ...
     return index;
   endfunction
```

5. THE SEQUENCE RESPONSE

So far we have focused on sequences running on a single sequencer. Now we turn to the interaction between sequencers and drivers, and to layered sequencers.

Layered sequencers are an important issue because they address the use case of modeling protocol stacks in the UVM environment. Almost by definition, because the driver is required to "wiggle the pins" of the DUT, a sequencer connected directly to a driver must generate transactions that represent the lowest level functional protocol used to communicate with the design-under-test. But many applications will require one protocol to be embedded within another protocol as we climb the protocol stack, and each layer would typically be represented in UVM by having sequences running on a distinct sequencer.

Higher level sequencers generate transactions which they send to lower level sequencers, which translate those transactions into other transaction types which they send in turn to even lower level sequencers and ultimately to drivers. In UVM, each of these transactions is known as a request. A UVM sequence running on a sequencer sends requests to a driver (or to a lower level sequencer). The issue then arises as to how to pass information back up the stack in the direction away from the driver.

There are two basic choices: either use the response that is built into the sequencer-driver interaction mechanism, or use analysis ports. The sequence response is appropriate when the response information is intrinsic to the protocol being modeled, such as when returning data as part of executing a *read* transaction. Analysis ports are appropriate when the response information can be separated from the protocol and carried as "side band" information. There are many cases where either technique could be used.

A consequence of using analysis ports to carry information up the stack, away from the DUT, is that analysis ports are non-blocking, so transactions must arrive at their destination in zero time. This is fine as long as it can be tolerated by the design of the verification environment, but does not permit the situation where a transaction has to be stalled while waiting for a higher level component to be ready. A sequence response, on the other hand, has to be properly synchronized with the request to which it corresponds, although UVM provides techniques to permit pipelined and out-of-order responses, as we will see below.

Let’s start with an example to review the basics. The sequencer-driver interface uses two transaction types, the request transaction and the response transaction, which may be the same or different. In the examples below we will keep them the same for simplicity:

```
class my_seq extends uvm_sequence #(my_tx);
  ...
  task body;
    ...
    // Create request, wait for driver, send to driver
    req = my_tx::type_id::create("req");

    // Put request into the sequencer          arbitration queue
    start_item(req);
    if( !req.randomize() ) ...
    finish_item(req);

    // Wait for response from driver (a blocking call)
    get_response(rsp);
    ...


class my_driver extends uvm_driver #(my_tx);
  ...
  task run_phase(uvm_phase phase);
    forever
    begin
      // Wait for request from sequence (a blocking call)
      seq_item_port.get_next_item(req);

      // Wiggle pins of DUT
      @(posedge dut_vi.clock);
      dut_vi.cmd <= req.cmd;
      ...

      // Create response transaction
      rsp = my_tx::type_id::create("rsp");
      rsp.data = dut_vi.data;
      rsp.set_id_info(req);

      // Send response back to sequence (goes into response queue)
      seq_item_port.item_done(rsp);
      ...
```

The protocol between the sequence (running on a sequencer) and the driver is:

1. Sequence and driver wait for each other to be ready
2. Sequence sends request to driver and waits for response
3. Driver gets request, processes request, copies transaction id information from the request to the response, and then sends the response back
4. Sequence receives the response

5.1. Pipelined responses

So far, so good, but a little experimentation will reveal some limitations to the approach used in the example above, the most fundamental of which is that, as things stand, the request and response cannot be pipelined. In particular, **get_next_item** cannot be called before the previous **item_done**, so the driver cannot start to process the next transaction before having sent the previous response. Also, vice versa, the sequence is waiting for the previous response before sending the next request. These issues can be addressed by sending the response using the **put** method instead of **item_done**. It will also be necessary to introduce concurrent processes in order to keep multiple request-response pairs in flight at the same time. For example, the sequence can fork processes to wait for pipelined responses:

```
// Create request, wait for driver, send to driver
req = my_tx::type_id::create("req");
start_item(req);
if( !req.randomize() ) ...
finish_item(req);

req_id[i] = req.get_transaction_id();

// Spawn a process to receive the response
fork
  begin
    int id = req_id[cnt++];
    get_response(rsp, id);
    ...
  end
join_none
```

The code fragment above requires a little explanation. Each transaction generated by a sequence is automatically allocated a transaction id by the call to **finish_item**. This id can be retrieved and then used to associate the request with the corresponding response by passing the id as the second argument to **get_response**, which will block until a response with the correct id appears in the response queue. Since each call to **get_response** is forked to run in a separate process, the responses can be pipelined and can even be sent out-of-order.

The driver can now process multiple requests concurrently by forking a separate process to handle each transaction:

```
forever
begin
  // Wait for request from sequence (a blocking call)
  seq_item_port.get(req);

  // Wiggle pins of DUT
  @(posedge dut_vi.clock);
  dut_vi.cmd <= req.cmd;
  ...
  fork
    begin
      my_tx resp;
      resp = my_tx::type_id::create("resp");
      resp.data = dut_vi.data;
      resp.set_id_info(req);

      // Consume some time before sending the response
      repeat(2) @(posedge dut_vi.clock);

      seq_item_port.put(resp);
    end
  join_none
end
```

Notice that the driver is calling **put** instead of **item_done** to send the response.

In SystemVerilog, you always have to be careful with the control flow around a **fork join_none** because the processes represented by each branch of the **fork** will not necessarily start to execute until the main process has yielded control. In the code fragment above, the line **resp.set_id_info(req)** will not get executed until the surrounding process has been blocked by the call to **get**. However, it will execute before **get** returns, so the correct **req** transaction always gets captured.

UVM offers an alternative way for the sequence to receive incoming responses: instead of calling **get_response**, a sequence can define a response handler method, as follows:

```
task body;
  use_response_handler(1);
  forever
    // Send request to driver
  ...
endtask

function void response_handler(
                    uvm_sequence_item response);
  $cast(rsp, response);
  id = rsp.get_transaction_id();
  // Process response
  ...
endfunction
```

The call use_response_handler(1) informs the sequence that the **response_handler** method is being overridden and is to be called for each incoming response. (The method names **use_response_handler** and **response_handler** are built into the UVM base class libaray.)

5.2. Passing responses through multiple layers

It is straightforward to extend this example to pass responses up though multiple sequencers representing the layers of a protocol stack. The practical problem is keeping clear in your mind that each connection from a sequencer to the sequencer or driver below requires its own distinct request and response transactions. Each request can only be associated with zero or one responses, and you have to decide whether it is to be zero or one when you design your sequence and driver classes. Each time a response is sent back up to the level above, you have to make sure that the correct id info is copied into the response object (by calling **set_id_info**).

Here is an example of a sequence that runs on a sequencer in the middle of a stack of sequencers, getting requests from a higher level sequencer and sending requests to a lower level sequencer or driver. Such a sequence is sometimes referred to as a *translation sequence* because it effectively translates between the protocols being modeled at the upper and lower layers of a protocol stack. As you study this example, bear in mind that many of the details are omitted, and that the relationship between lower and upper layer transactions could be one-to-many, many-to-one, or many-to-many: that is why the example shows some indexed names such as **req_up[j]**, where **req_up** is an array of transactions pulled down from the upper layer sequencer. The sequence contains a variable **seqr_upper** that must be set to point to the higher level sequencer before the sequence shown below is started on its own sequencer:

```
class lower_seq extends uvm_sequence #(my_tx);
  ...
  // Reference to upper-layer sequencer
  my_sequencer seqr_upper;

  task body;
    ...
    // Get request(s) from upper layer sequencer
    seqr_upper.get(req_up[j]);
    ...

    // Create request(s) and send to lower layer
    req = my_tx::type_id::create("req");
    start_item(req);
    if( !req.randomize() ) ...
    finish_item(req);

    // Store id of request(s) to match with response later
    req_id_lower[i] = req.get_transaction_id();

    // Fork a process to receive the response(s) from the lower layer
    fork
      begin
        int id = req_id_lower[cnt++];
        get_response(rsp, id);
        ...
        // Send response(s) back up the stack
        my_tx rsp_up= my_tx::type_id::create("");
        rsp_up.data = rsp.data;
        rsp_up.set_id_info(req_up[j]);
        seqr_upper.put(rsp_up);
      end
    join_none
    ...
```

6. MULTIPLE SEQUENCER STACKS

Synchronizing the behavior of multiple agents or scoreboards, either vertically within a single sequencer stack or horizontally across multiple sequencer stacks, is a very common issue and potentially a very difficult problem to handle in UVM. The default way to tackle this issue should be to synchronize the UVM drivers to the clocks, strobes, and other low-level synchronization signals in the DUT interface, and as far as possible have all the higher level components in the UVM verification environment, including sequencers, scoreboards and checkers, respond immediately without blocking and without delay. In other words, the drivers are synchronized to clocks in the DUT interface and pull down transactions from a stack of sequencers, which are always able to respond immediately on-demand. Keeping timing and synchronization confined to the driver layer in this way simplifies the problem enormously, but unfortunately this approach is not always possible. It is always best to have coverage and checking performed in a non-blocking manner by sending transactions from monitors using analysis ports. However, it is sometimes necessary to keep one sequencer stack (that feeds transactions into a one DUT interface) idling until another parallel sequencer stack is ready to proceed, perhaps because it was waiting for a response from another DUT interface.

When it comes to ad hoc communication and synchronization between components, UVM offers several options aside from the sequencer-driver interface, virtual sequences, and analysis ports. There are blocking and non-blocking transaction-level interfaces (ports and exports), there are events and barriers, and there are callbacks. Each has its own advantages and disadvantages, and there are usually several different solutions that can be made to work, the choice being made according to individual or corporate taste.

If a sequencer is unable to provide the next transaction immediately, then the component below (usually the driver) may need to take some alternative action. In general, having a driver blocked waiting for a sequencer would be a bad idea:

```
seq_item_port.get(req);  // Had better not block!
@(posedge dut_vi.clk);
```

This issue can be addressed directly by using the non-blocking version of the sequencer interface, as follows:

```
seq_item_port.try_next_item(req);
if (req == null)
begin
  // Wiggle pins of DUT to represent an idle cycle
  dut_vi.idle <= 1;
  ...
  @(posedge dut_vi.clock);
end
else
```

```
begin
  // Must be called in same time step that try_next_item returns non-null
  seq_item_port.item_done();

  // Wiggle pins of DUT for regular transaction
  dut_vi.idle <= 0;
  ...
  @(posedge dut_vi.clock);
  ...
  seq_item_port.put(resp);
```

Method **try_next_item** always returns immediately. You should then test the return value, a **null** reference meaning that the next item is not yet ready. On the other hand, if **try_next_item** does return a valid transaction reference then the **item_done** method must be called in the same time step, though this does not you prevent you from calling the **put** method to return a response some time later if required, so is not an obstacle to modeling pipelined transactions.

Calls to **try_next_item** can be stacked. That is, every sequence in a stack can call **try_next_item** to pull down a request from the sequencer above it in the stack. The only practical pitfall is to ensure that the matching **item_done** calls occur in the same timestep.

Since the driver can now tolerate the sequencer not being ready, we could have a sequence somewhere further up the stack waiting for an external event. In this example we will make use of the **uvm_event** to provide synchronization between two parallel sequencer stacks:

```
task body;
  my_tx tx;
  uvm_event_pool pool =
              uvm_event_pool::get_global_pool();

  // Find event in event pool, identified by name
  uvm_event sync_event = pool.get("sync_event");

  // Wait for the event to be triggered (a blocking call)
  sync_event.wait_trigger();

  // Retrieve a transaction that was passed along with the event
  $cast(tx, sync_event.get_trigger_data());

  // Create request, wait for driver, send to driver
  req = my_tx::type_id::create("req");
  start_item(req);
  ...
```

In general, the event could be notified from anywhere in the UVM environment. In this example, the event is notified when a sequence running on a parallel sequencer stack receives a response notification:

```
class ano_sequence extends uvm_sequence #(my_tx);
 ...
 function void response_handler(
                   uvm_sequence_item response);
   uvm_event_pool ev_pool =
              uvm_event_pool::get_global_pool();
   uvm_event sync_event = pool.get("sync_event");
   $cast(rsp, response);
   sync_event.trigger(rsp);
 endfunction

endclass
```

What we are modeling here is ad hoc horizontal communication between sequencer stacks. Generally we will try to push all the detailed timing and synchronization down to the bottom of the stack where the lowest level interfaces are modeled in the drivers. However, on the occasions when a higher level sequencer needs to block and wait for some other process to catch up, we can used a **try_next_item** call from the driver to handle the situation where the stack is stalled and then have the driver generate idle cycles or background traffic, assuming our application and test cases can handle such an approach.

7. THE CONFIGURATION DATABASE

The configuration database is best thought of as a general repository for information that can be used to parameterize the UVM environment during the build phase and then remains available for access during the later phases. All parameter values stored in the configuration database are associated with specific paths in the UVM component hierarchy. In other words, the configuration database can be used to set parameters on specific UVM components.

The current UVM configuration database is an evolution of the configuration interface from OVM. Unfortunately this legacy has caused a few pitfalls to be left around for the unwary. In the move from the OVM configuration interface (set_config*/get_config* methods) to UVM, the configuration mechanism has been structured into two distinct layers. The OVM configuration interface of the OVM component is mimicked by the UVM configuration database, which is in effect a convenience interface on top of the UVM resource database. In other words, any parameters written into the configuration database are actually stored in the resource database, and may be accessed directly through the methods of the resource database.

The resource database stores records consisting of a scope, a name, a value, and some secondary attributes that can usually be ignored. The scope and the name are both text strings. The classes uvm_resource_db and uvm_config_db are parameterized with the type of the parameter value such that values of any type can be stored directly without needing to be wrapped as uvm_objects. This includes built-in types, user-defined types, and even virtual interfaces. The intent of the scope is to help reduce the probability of name collisions between parameters used by unrelated verification components.

The configuration database provides a layer on top of the resource database that uses the scope stored in each record of the resource database to represent a path in the UVM component hierarchy. Significantly, the methods of the configuration database allow paths to include wildcards such that a single record in the configuration database can be used to set parameters that apply to multiple components.

The basic calls used to access the configuration database are as follows:

```
uvm_config_db#(T)::set(
                 caller, "path", "name", value);
...
uvm_config_db#(T)::get(
                 caller, "path", "name", value);
```

One of the most important features of the resource-cum-configuration database is that the information it stores is linked to the UVM verification environment only through the text strings used to define the scope and name of each parameter and not by any direct object references. In other words, it is entirely possible to set and get parameters in the configuration database using "fictitious" scopes and names. More practically, it is possible to set parameters in the configuration database prior to the construction of the UVM component hierarchy to which they will apply. In fact, this is one of the main use cases for the configuration database: to set parameters in advance that are then used to control the construction of the component hierarchy during the build phase.

The other thing you need to understand about the OVM configuration interface, mimicked by the UVM configuration database, is what happens in the situation where several different components attempt to set the value of one-and-the-same configuration parameter. In OVM, where the methods of the configuration interface belong to ovm_component, there is a clearly-defined search order used when retrieving parameter values by calling get_config_*: in cases where both the path name and the parameter name match, calls to set_config_* made from closer to the root of the component hierarchy take precedence over calls made closer to the leaves of the component hierarchy. In other words, if you set the same parameter on the same component from several different locations in the component hierarchy, calls made from lower down the hierarchy get overridden by calls made from further up the hierarchy, with set_config_* calls not associated with any component winning over calls made from within the hierarchy. This OVM mechanism is mimicked in UVM by having the UVM resource database store a queue of resources in order of creation in cases where there are several resources with the same name and scope, and having the **get** method return the match at the front of the queue.

The UVM resource database actually goes a little further than this. It is possible to change the default precedence associated with each resource, which in effect alters the parameter value that gets returned where there exist multiple parameters with the same name and scope. However, this feature is not exposed by either the uvm_config_db or the uvm_resource_db classes: to modify the precedence of resources you have to dive deep into the classes that underlie the resource database, which for most users is probably not worth the effort.

7.1. Configuration database pitfalls

Having set the scene by reviewing some of the mechanism of the configuration database, we can now review some of the pitfalls. The first pitfall is that uvm_config_db only mimics the behavior of OVM correctly when the first argument to set/get, the caller, is used in a certain conventional way. To get the desired effect, a reference to the calling component must be passed as the first argument (e.g. **this**), with **null** being passed only in the case that **set** is called from outside the UVM component hierarchy, e.g. from a SystemVerilog module. A call such as

```
uvm_config_db#(T)::set(null, "*.m_seqr", ...);
```

when made from within a UVM component would correctly interpret the path name, with its wildcards, but would in effect jump to the front of the queue so that it would take precedence over any other matching calls that have a reference to a UVM component passed as the first argument. In other words, with respect to the search order it is as if this call had been made from a SystemVerilog module, not from a UVM component.

The second pitfall is that uvm_config_db::set only mimics the OVM precedence when called *during the build phase*. Thereafter it is a case that the most recent call always becomes the winner. So a call such as

```
uvm_config_db#(T)::set(this, "path", "name", v);
```

made from deep within the UVM component hierarchy during the run phase will in effect take precedence over any matching calls made during the build phase.

The third pitfall concerns the **get** method. The second argument is a path name, expressed relative to the component passed as the first argument. If the first argument is **null**, then the second argument would represent an absolute path name relative to the top of the UVM component hierarchy. So the call

```
uvm_config_db#(T)::get(null, "*.m_seqr", ...);
```

should theoretically match any path that ends with the characters ".m_seqr". But this *does not work*, due to a feature/bug of the UVM code base. Having **null** as the first argument prevents a proper wildcard lookup. This can be worked around by separating the wildcard lookup from the **get** call, as follows:

```
uvm_component comp = uvm_top.find("*.m_seqr");
if (uvm_config_db#(T)::get(null,
                    comp.get_full_name(), ...);
```

To help avoid the above pitfalls, it is best practice to always pass a component reference as the first argument to **set** and **get** except in the one special case of calling **set** from a SystemVerilog module prior to calling **uvm_top.run_test**, in which case the first argument has to be **null**.

This brings us to the issue of accessing the configuration database from sequences. In OVM this was not at all straightforward, because the configuration interface consisted of methods of class ovm_component, and a sequence is not a component. In UVM you *can* call uvm_config_db::**get** directly from a sequence, but you must be sure to avoid the pitfalls highlighted above. So given that in general it is best practice to pass a component reference as the first argument to **get**, we will want to set/get configuration parameters using locations in the component hierarchy, even when using those parameters in sequences. The most straightforward solution is to make use of the **p_sequencer** variable, as described above:

```
class my_sequence extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_sequence)
  `uvm_declare_p_sequencer(my_sequencer)
  ...
  if (uvm_config_db#(T)::get(p_sequencer, "", ...
```

The macro declares a variable **p_sequencer** of the type passed as a macro argument, and sets that variable to point to the sequencer on which the sequence is running.

7.2. Modifying parameters at run time

As noted above, the values of configuration parameters can be modified during the run phase, even though the hierarchical search rules used during the build phase no longer apply. In OVM, the problem with pushing through parameter changes during the run phase was the lack of a mechanism to make this work without having to constantly poll the parameters for changes. In UVM there is a simple mechanism to alert verification components to parameter changes in the configuration database: the **wait_modified** method.

Before showing the details, there is one important caveat that needs to be mentioned. The uvm_config_db **set** and **get** methods are computationally quite expensive, so these are not methods you would want to call in the inner loop of your SystemVerilog program. In other words, you should not make frequent calls to **set** and **get** during the UVM run phase: it is usually considered best practice to restrict most calls to **set** and **get** to the build phase.

With that caveat in mind, any UVM component that needs to know about parameter value changes can fork off a concurrent process that sits and waits for a change to a given parameter, as follows:

```
task run_phase(uvm_phase phase);
  ...
  fork
    forever begin
      int numb;
      uvm_config_db#(int)::wait_modified(
                     this, "", "number");

      void'(uvm_config_db #(int)::get(
                     this, "", "number", numb));
      ...
    end
  join_none
```

wait_modified is a blocking method that waits for a given parameter to be modified. The parameter is identified using exactly the same scope rules as for the **get** method.

It is quite possible that a single call to **set** may cause several waiting processes to awake if the **set** call uses wildcards in the path.

8. CONCLUSION

UVM is a surprisingly rich class library that conceals many interesting features as well as pitfalls for the unwary. The official documentation [2, 3] is excellent, and yet there is sufficient complexity in the library that the documentation itself leaves a lot unsaid. In this paper we have explored just some of the features that are worthy of further attention. You are encouraged to explore the UVM codebase yourself to discover what works and what does not by writing and running examples. At the same time, you should keep an eye on what others are saying about best practice by visiting some of the websites mentioned below [4, 5, 6, 7].

9. REFERENCES

[1] IEEE Std 1800-2009 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language", http://dx.doi.org/10.1109/IEEESTD.2009.5354441

[2] Universal Verification Methodology (UVM) 1.1 Class Reference, updated September 19, 2012

[3] Universal Verification Methodology (UVM) 1.1 User’s Guide, May 18, 2011

[4] On-line resources from http://www.accellera.org/downloads/standards/uvm

[5] On-line resources from http://www.uvmworld.org/

[6] On-line resources from http://www.doulos.com/knowhow/sysverilog/uvm/

[7] On-line resources from https://verificationacademy.com/